# ES5 & ES2015

## ES5 getters and setters

- **get** is a property that enables you to dynamically retrieve the value of properties based on the internal state of the object

- **set** is the opposite and enables you to dynamically set the state of an internal property for an object.

Array's have far more useful helper methods and by using Object.keys instead of the traditional for (var key in obj) syntax it is much easier to chain transformational methods to the object in the same way we would do with an array.

let obj = {

firstName: 'Tom',

lastName: 'Adams',

get fullName() {

return this.firstName + ' ' + this.lastName;

},

set fullName(value) {

let nameArray = value.split(' ');

this.firstName = nameArray[0];

this.lastName = nameArray[1];

}

}

Object.keys is a function on the Object constructor that takes in an object and outputs that objects keys into an array.

ES2015 introduces two new ways to declare variables that help to get around some unexpected behavior that can occur with variable assignment and variable hoisting in Javascript. `let` and `const` create a new scope when they appear inside of curly braces { }

## Var, let, const

`**let**` is just like `**var**` except it is blocked scoped, which prevents variables from being hoisted to the top of our scope during the javascript declaration, which is what occurs with the `var` keyword.

`const` is just like `let` except for it cannot be reassigned after it has been initialized with a value. It is also block scoped.

Blocked scope variables declarations to behave similar to most other languages. When using ES2015 you should *always use `let` and `const` instead of var*.

Arrow functions are new ES2015 syntax that makes it more straightforward to maintain the parent object scope inside of a callback or method. ES5 solutions for the issue of losing parent scope are to use .bind(this) on the callback function or to create a copy of this and reference the copy inside of the callback function.

Template strings are the use of the back-tick symbol (`) to open and close a multi-line string. Template strings also allow for string interpolation using the ${ data } syntax.

let myData = {

data: 'Hello'

};

let template = `

<div>

${ myData.data }

</div>

`;

## Destructuring

Destructuring is a simple syntax for dynamiclly plucking properties off of a data structure while assigning them to distinct variables.

You can destructure a data structure by referencing an object's property names inside of curly braces.

var object = { "a": 1, "b": 2 }

var {a, b} = object;

console.log(a, " | ", b);

=> 1 | 2

In modern frameworks like Angular that are built on TypeScript (which is a superset of ES2015), destructuring is common in import statements to pluck specific modules from an exported file.

let prism = {

length: 5,

width: 7

};

function calculateArea({length, width, height = 10}) {

return length \* width \* height;

}

console.log(calculateArea(prism));

## Rest arguments

If the last named argument is prefixed with (...), the argument collects itself and all consecutive arguments. The arguments are converted into an array.

printObjects('prefix', {name: 'Tom'}, {name: 'Andy'}, {name: 'Dan'});

function printObjects(prefix, ...args) {

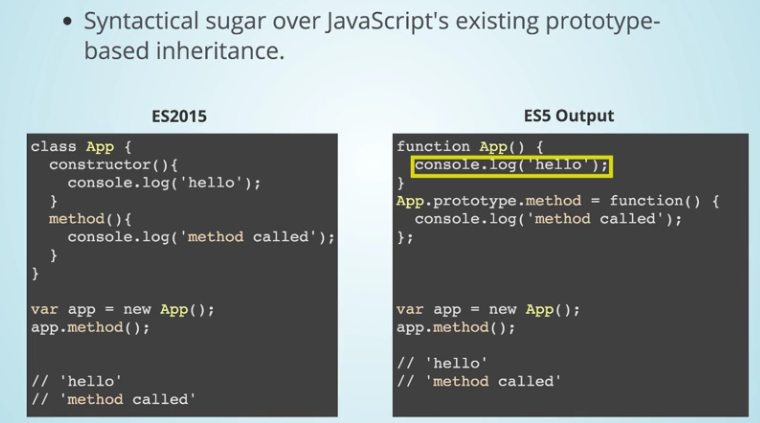
args.forEach((arg) =>

console.log(prefix, arg.name)

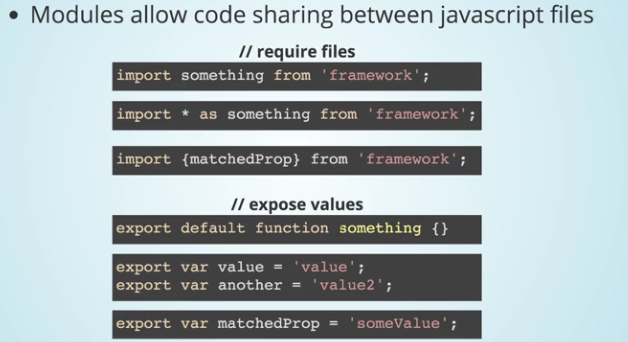
);

}

## Classes



## Modules



## TS Decorators

function clean(target) {

target.cleaned = true;

}

function clean2(value: boolean) {

return function(target) {

target.cleaned2 = value;

}

}

@clean2(false)

@clean

class Animal {

}

let animal = new Animal();

console.log(animal);

## Observables

<body>

<script src="https://unpkg.com/rxjs@5.2.0/bundles/Rx.min.js"></script>

<input type="text">

</body>

………………….

var $ = document.querySelector.bind(document);

var myInput = $('input');

var obs = Rx.Observable.fromEvent(myInput, 'input');

obs

.map(function (event) {

return 'Value: ' + event.target.value;

})

.subscribe(function(value) {

console.log(value);

});